Reporte Compilador de Cool

Carlos Jorge Rodrigues Cuello C411 @CarlosJorgeR

Roberto Zahuis Benitez C411@Zahuis314

Sándor Martín Leyva C412 @SandorMLyeva

Arquitectura del compilador

Lexer y Parser

Para la implementación del lexer y el parser se utilizó como herramienta ANTLR4 donde en `COOLgrammar.g4` se definió el lexer dando una definición de cada token a partir del uso de expresiones de regulares. También en dicho fichero se definió la estructura de la gramática, en cuanto a este aspecto cabe destacar que dicho ¨framework¨ maneja por nosotros la recursión izquierda inmediata eliminando la ambigüedad que introduce esta y establece el orden de las operaciones acorde al orden en que se definen las producciones de cada no terminal lo cual simplificó en gran medida esta parte del proyecto. El árbol de derivaciones generado por el parser esta definido en `COOLgrammarParser.cs` el cual se usó como el **AST** de Cool.

Análisis semántico

En este se visita el **AST** en 3 pasadas:

1. Para capturar la definición de los tipos.
2. Para capturar la definición de los atributos y métodos. En esta pasada las clases se analizan en orden topológico para cuando se analiza un método de una clase A heredado de otra clase B tener los métodos de dicha clase B previamente analizados.
3. Para analizar las expresiones definidas en el cuerpo de los métodos y las inicializaciones de los atributos.

En estas 3 pasadas se usaron estructuras que implementan `IContext` las cuales se usan para:

- Atrapar las definiciones de los **tipos, atributos**, **metodos**, **parámetros**, **variables** introducidas por las expresiones let.
- Preguntar si en un determinado contexto alguna de estas definiciones existe.
- Acceder a estas definiciones y preguntar por sus propiedades.

Generación de código

Generando de código de Cool a Cil

En este de forma parecida al análisis sintáctico se hacen dos pasadas:

1. Para generar el código de los tipos y definir las funciones. Esta pasada en nuestro proyecto se hace indispensable para más adelante en las funciones tener acceso a las definiciones de todos los tipos y las funciones de `Cil`.

2. Para generar el código de `3-direcciones` de las funciones.

Cabe destacar en el **AST** de **Cil** tiene la siguiente estructura:

- Los tipos están definidos por los nombres de los atributos `IAttributeCil` y los métodos `IFunctionTypeCil`; estos últimos compuestos por el nombre virtual del método *(donde todo tipo que conforma al actual posee el mismo nombre virtual de dicho método)* y el nombre real del método `IFunctionCil` *(el cual es el nombre real del método que usa el tipo, definido en la zona de las funciones de CIL)*
- La zona donde se guardan las variables que se les asigna las cadenas de texto que serán usada por las funciones `DataStringCil`.
- La zona donde están definidas las funciones de todo el programa `IFunctionCil` las cuales están compuesta por la definición de los parámetros que recibe, las variables locales que se usan dentro de esta y un bloque de instrucciones de 3 direcciones.

Generando de código de Cil a Mips

El programa en mips se divide en 2 secciones fundamentales:

.data en la cual van las definiciones de tipos, variablesglobales y strings

.text que contiene todo el código del programa.

Ambas secciones se calculan en la clase `CilToMips`. Esta recorre todos los tipos definidos en Cil y escribe sus homólogos en **.data** con la siguiente estructura.

```
type_<nombreDelTipo>:
#   Cantidad de atributos del tipo
type_<nombreDelTipo>_Length: .byte 0
#   Puntero a la definicion de la función
type_<nombreDelTipo>$<nombreDelMetodo>: .word <nombreDelTipo>$<nombreDelMetodo>
    .
    .   # Punteros a las direcciones de memoria de la implementación de las funciones
    .
#   Cantidad de funciones
type_<nombreDelTipo>_count_methods: .word 1
#   Direccion al tipo padre para facilitar el trabajo con la herencia
type_<nombreDelTipo>_parent: .word 0
#   String del nombre del tipo
type_<nombreDelTipo>_name: .asciiz "<nombreDelMetodo>"
```

En la sección **.text** la primera definición que se va a encontrar es el método entry, el cual va a ser el punto de entrada del programa. Después le siguen las definiciones de las funciones restantes.

Su estructura consiste en:

```
#   etiqueta para identificar la definicion
<nombreDelTipo>$<nombreDeLaFuncion>:
#   se mueve sp para reservar el espacio de las variables que se van a utilizar en la función y
#   no afectar a la funcion que la llame
sub $sp, $sp, <(cantidadDeParametros+cantidadDeVariablesLocales+1)*4>
#   se guarda la posición de retorno, para evitar problemas con llamados a otras funciones en el
#   interior
sw $ra, 4($sp)
    .
    .   # Código del cuerpo de la función
    .
#   se carga la posición de retorno
lw $ra, 4($sp)
#   se pone el resultado calculado en el registro de retorno
move $v0, <Registro>
#   se deja sp en la posición que estaba antes del llamado de la función
addi $sp, $sp, <(cantidadDeParametros+cantidadDeVariablesLocales+1)*4>
#   se salta a la posición de retorno
jr $ra
```

Otro punto muy importante es la forma de reservar espacio en memoria para llevar a cabo la instancia de un objeto. Como se conoce, esta es a partir de la palabra reservada ALLOCATE. En la sección **.data** se tiene reservado un `heap` para almacenar las instancias de los objetos y `heapPointer` que tiene guardado la primera posición libre del heap.

```
#   se pone en $t1 la cantidad de espacio que hay que reservar ,el +2 es debido a que además de
todos los atributos del objeto, también se guarda el nombre del tipo, y la cantidad de atributos
addi t1, t1, <(cantidadAtributosDelTipo+2)*4>
sw $t1, heapPointer
la $t0, heap
add $t0, $t0, $t1
move $v0, $t0
la $t2, type_{type.Name}_name
sw $t2, ($t0)
add $t0, $t0, 4
#   Escribir la cantidad de bytes de los argumentos
li $t2, {type.Attributes.Count}
sw $t2, ($t0)
```

Uso del Compilador

Para el uso del compilador solo debe correr

```
<nombre>.exe <programa>.cl
```

El cual genera el un `<nombre>.asm` listo para correr en spim ;)